# 5 Building a simple OAuth authorization server

***This chapter covers***

- Managing registered OAuth clients
- Having a user authorize a client
- Issuing a token to an authorized client
- Issuing and responding to a refresh token

In the last two chapters, we built an OAuth client application that fetched a token from an authorization server and used that token at a protected resource, and we built the protected resource for the client to access. In this chapter, we'll build a simple authorization server that supports the authorization code grant type. This component manages clients, performs the delegation action core to OAuth, and issues tokens to clients.

> **NOTE** All of the exercises and examples in this book are built using Node.js and JavaScript. Each exercise consists of several components designed to run on a single system accessible from *localhost* on various ports. For more information about the framework and its structure, see appendix A.

The authorization server is arguably the most complex component in the OAuth ecosystem, and it is the central security authority throughout a given OAuth system. Only the authorization server can authenticate users, register clients, and issue tokens. During the development of the OAuth 2.0 specifications, wherever possible

complexity was pushed onto the authorization server from the client or the protected resource. This is largely due to the arity of the components: there are many more clients than protected resources, and many more protected resources than authorization servers.

We'll start out by building a simple server in this chapter, and we'll be building in more capability and functionality as we go.

## 5.1 Managing OAuth client registrations

In order for clients to talk to the OAuth server, the OAuth server needs to assign a unique client identifier to each client. Our server is going to use static registration (we'll cover dynamic client registration in chapter 12), and we're going to store all of our client information in a server variable.

Open up `ch-5-ex-1` and edit the `authorizationServer.js` file. We won't be editing the other files in this exercise. At the top of the file, there's an array variable that stores client information.

```
var clients = [

];
```

This variable, currently empty, will act as the data store for all client information at the server. When part of the server needs to look up information about a client, it will look in this array to find it. In a production OAuth system, this type of data is usually stored in a database of some kind, but in our exercises we wanted you to be able to see and manipulate it directly. We're using an array here because it's assumed that an authorization server will be handling many clients in an OAuth system.

> **Who creates the client ID?**
>
> We've already configured the client in `client.js` to expect a certain ID and secret, so we'll be copying those here. In a regular OAuth system, the authorization server issues the client ID and client secret to the client software, just like we did in our last exercise. We've done things this way to save you some typing and to confine the exercise to editing a single file wherever possible. However, if you want, you can open up client.js and tweak the values in its own configuration.

First, let's take in the values from the client itself, values that the authorization server doesn't generate. Our client's redirect URI is `http://localhost:9000/callback`, so we'll create a new object in our list of clients that has that:

```
var clients = [
   {
        "redirect_uris": ["http://localhost:9000/callback"]
   }
];
```

Next, we need to assign the client an ID and secret. We'll give it the same values that we used in the exercises in the last chapter, `oauth-client-1` and `oauth-client-secret-1`, respectively. (The client has already been configured with this information.) This fills out our object to the following structure:

```
var clients = [
   {
        "client_id": "oauth-client-1",
        "client_secret": "oauth-client-secret-1",
        "redirect_uris": ["http://localhost:9000/callback"],

   }
];
```

Finally, we'll need a way to look up this information based on the client ID. Where a database would generally use a query, we've supplied a simple helper function to dig through our data structure to find the right client.

```
var getClient = function(clientId) {
   return __.find(clients, function(client) { return client.client_id ==
   clientId; });
};
```

The details of this function's implementation aren't important, but it's doing a simple linear search across the list for the client with the given client ID. Calling this function gives us back the client we asked for as an object, or `undefined` if the client wasn't found. Now that our server knows about at least one client, we can start putting together the code that acts for the server's various endpoints.

## 5.2 Authorizing a client

The authorization server is required to have two endpoints in the OAuth protocol: the authorization endpoint, which serves front-channel interactions, and the token endpoint, which serves back-channel interactions. If you want the details of how the front channel and back channel work, and why both are needed, see the explanation in chapter 2. In this section, we'll be building the authorization endpoint.

**Does it have to be a web server?**

In a word, yes. As we covered in chapter 2, OAuth 2.0 is designed as an HTTP-based protocol. In particular, the authorization server is expected to be available to the client over HTTP for both the front and back channels. The authorization code grant type, which we're using in our example, requires both the front-channel and back-channel interfaces to be available. The front channel needs to be reachable by the resource owner's browser, and the back channel needs to be reachable directly by the client itself. As we'll see in chapter 6, there are other grant types in OAuth that use only the front channel or only the back channel; our exercises here are going to use both.

***(continued)***

There are some ongoing efforts to port OAuth to non-HTTP protocols such as Constrained Application Protocol (CoAP), but they're still based on the HTTP-bound original specification and we won't be covering those directly here. For an added challenge, take the HTTP servers in the exercises and port them to another carrier protocol.

### 5.2.1 The authorization endpoint

The first stop that a user makes in the OAuth delegation process is the authorization endpoint. The authorization endpoint is a front-channel endpoint to which the client sends the user's browser in order to request authorization. This request is always a GET, and we'll be serving it up on `/authorize`.

```
app.get("/authorize", function(req, res){

});
```

First, we need to figure out which client is making the request. The client passes its identifier in the `client_id` parameter, so we can pull that out and look up the client using our helper function from the last section:

```
  var client = getClient(req.query.client_id);
```

Next, we need to check whether or not the client passed in exists. If it doesn't, we can't authorize it to access anything, so we display an error to the user. The framework includes a simple error page that we're using here to show the error to the user.

```
if (!client) {
  res.render('error', {error: 'Unknown client'});
  return;
```

Now that we know which client claims to be asking, we need to do some sanity checks on the request. At this point, the only thing that's been passed in through the browser is the `client_id`, and since it gets passed through the front channel in the browser, this is considered public information. At this point, anybody could be pretending to be this client, but we've got some things that can help us make sure it's a legitimate request, chief among them checking the `redirect_uri` that was passed in against the one that's been registered by the client. If they don't match, this is also an error.

```
} else if (!__.contains(client.redirect_uris, req.query.redirect_uri)) {
  res.render('error', {error: 'Invalid redirect URI'});
  return;
```

The OAuth specification, and our simple implementation of it, allows for multiple `redirect_uri` values for a single client registration. This can help with applications that might be served on different URLs in different circumstances, allowing the consents to be tied together. As an additional exercise, once this server is up and running, come back to it and add support for multiple redirect URIs.

OAuth defines a mechanism for returning errors to the client by appending error codes to the client's redirect URI, but neither of these error conditions do so. Why is that? If either the client ID that's passed in is invalid or the redirect URI doesn't match what's expected, this could be an indicator of an attack against the user by a malicious party. Since the content of the redirect URI is completely out of the control of the authorization server, it could contain a phishing page or a malware download. The authorization server can never fully protect users from malicious client applications, but it can at least filter out some classes of attacks with little effort. For more discussion of this, see chapter 9.

Finally, if our client passes muster, we need to render a page to ask the user for authorization. The user will need to interact with this page and submit it back to the server, which will be served by another HTTP request from the browser. We're going to hang on to the query parameters from the currently incoming request and save them in the `requests` variable under a randomized key so that we can get them back after the form is submitted.

```
var reqid = randomstring.generate(8);
requests[reqid] = req.query;
```

In a production system, you can use the session or another server-side storage mechanism to hold this. The exercise includes an authorization page in the file approve.html, which we'll now render for the user. We pass in the client information that we looked up as well as the randomized key from earlier.

```
res.render('approve', {client: client, reqid: reqid});
```

The client information gets displayed to the user to aid them in making their authorization decision, and the randomized `reqid` key gets put into the form as a hidden value. This randomized value offers some simple cross-site request forgery protection for our authorization page, since we'll need it to look up the original request data in the next step for further processing.

All together, our function now looks like listing 7 in appendix B.

This is only the first half of processing a request to the authorization endpoint. Now it's time to prompt the resource owner and ask them to authorize the client.

### 5.2.2 Authorizing the client

If you've completed the previous steps, you can now run the code in its current state. Remember to run all three services: `client.js`, `authorizationServer.js`, and `protectedResource.js`. Starting at the OAuth client's homepage of `http://localhost:9000/` and clicking the "Get Token" button, you should be presented with the approval page, as shown in figure 5.1.

Our approval page is simple. It displays information about the client and asks the user for a simple approve/deny response. Now it's time to process the results of this form.

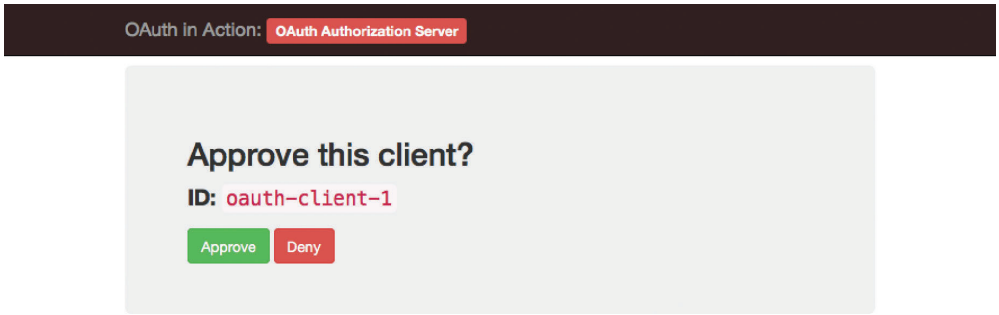


**Figure 5.1 A simple approval page**

### Who is the user, anyway?

In our exercises, we're leaving out one key step: authenticating the resource owner. Many methods can be used to authenticate the user, with lots of middleware capable of handling most of the heavy lifting. In a production environment, this is a vital step that will require careful implementation in order to be handled properly. The OAuth protocol doesn't specify or even care how the resource owner is authenticated, so long as the authorization server performs this step.

Try adding user authentication to the authorization and consent pages as an added exercise. You could even use an OAuth-based authentication protocol such as OpenID Connect (discussed in chapter 13) to log the resource owner in to the authorization server.

Even though the particulars of the form on this page are entirely specific to our application and not part of the OAuth protocol, using an authorization form is a fairly common pattern among authorization servers. Our form is set up to send an HTTP POST request to the `/approve` URL of the authorization server, so we'll start by setting up a listener for that.

```
app.post('/approve', function(req, res) {

});
```

When the form is submitted, it sends a request like the following, with the form values in HTTP form-encoded format:

```
POST /approve HTTP/1.1
Host: localhost:9001
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:39.0)
Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Referer: http://localhost:9001/authorize?response_type=code&scope=foo&client_
id=oauth-client-1&redirect_uri=http%3A%2F%2Flocalhost%3A9000%2Fcallback&
state=GKckoHfwMHIjCpEwXchXvsGFlPOS266u
Connection: keep-alive

reqid=tKVUYQSM&approve=Approve
```

Where did that `reqid` come from? Inside the page's HTML, the server has inserted the randomized string that we generated in the last step and passed to the template. In the rendered HTML, it looks something like this:

```
<input type="hidden" value="tKVUYQSM" name="reqid">
```

This value is submitted with the form, and we can pull that value out of the request body to look up the pending authorization request based on it. If we don't find a pending request for this code, it's possibly a cross-site forgery attack and we can send the user to an error page.

```
var reqid = req.body.reqid;
var query = requests[reqid];
delete requests[reqid];

if (!query) {
   res.render('error', {error: 'No matching authorization request'});
   return;
}
```

The next thing that we need to do is determine whether the user clicked the Approve or the. We can tell this by the presence of the `approve` variable in the form's submission, which is only included if the Approve button is clicked. A similar `deny` variable is sent by the Deny button, but we're treating anything other than the Approve button being clicked as a denial.

```
if (req.body.approve) {                    <-- User approved access

} else {                                   <-- User denied access

}
```

We'll handle the second case first because it's simpler. If the user denied access to an otherwise valid client, we can safely tell the client what happened. Since this is using front-channel communication, we don't have a way to send a message directly to the client. We can, however, use the same technique that the client used to send the request to us: take a URL hosted by the client, add a few special query parameters to that URL, and redirect the user's browser to the resulting location. The client's redirect URI is used for this purpose, and this is why we checked it against the registered client information when the first request arrived. In this case, we're sending back an error message telling the client that the user has denied access.

```
var urlParsed = buildUrl(query.redirect_uri, {
   error: 'access_denied'
});
res.redirect(urlParsed);
return;
```

If, conversely, the user approved the application, we need to first see what kind of response the client is asking for. Since we're implementing the authorization code grant type, we're going to look for the `response_type` value to be set to `code`. For any

other value, we return an error to the client using the same technique we just used. (We'll cover implementing support for other values in chapter 6.)

```
if (query.response_type == 'code') {
} else {
   var urlParsed = buildUrl(query.redirect_uri, {
        error: 'unsupported_response_type'
   });
   res.redirect(urlParsed);
   return;
}
```

**Handle the authorization code grant type here (see what follows for details)**

Now that we know what kind of response we're doing, we can generate an authorization code to send back to the client. We also need to store the code somewhere on our server so that we can look it up when the client comes back to the token endpoint in the next step. For our simple authorization server, we're going to use the same technique that we did when setting up the approval page and save it into an object on the server indexed by the authorization code we just generated. In a production server, this will likely be stored in a database, but it still needs to be accessible by the authorization code value as we'll see in a moment.

```
var code = randomstring.generate(8);

codes[code] = { request: query };

var urlParsed = buildUrl(query.redirect_uri, {
   code: code,
   state: query.state
});
res.redirect(urlParsed);
return;
```

Notice that we're not sending back just the `code`. Remember in the last chapter when we set up the client to pass a `state` parameter to the server for the client's own protection? Now that we're on the other end of the transaction, we need to pass through the `state` parameter exactly as it was sent to us. Even though clients aren't required to send the `state` value, the server is always required to send it back if one was sent in. So all together, our function to handle the response from the user approval page now looks like listing 8 in appendix B.

From here, the authorization server has handed control back to the client application and needs to wait for the next part of the transaction, a request to the token endpoint on the back channel.

## 5.3 Issuing a token

Back at the client, the authorization code generated in the previous section comes in through the client's redirect URI. The client then takes this code and creates a POST message to the authorization server's token endpoint. This backchannel communication happens outside the user's browser, directly between the client

and the authorization server. Since the token endpoint isn't user facing, it doesn't use the HTML templating system at all. Errors are communicated back to the client through a combination of HTTP error codes and JSON objects, which we'll see in use here.

We're going to set up a listener for POST requests on `/token` to handle this:

```
app.post("/token", function(req, res){

});
```

### 5.3.1 Authenticating the client

First, we need to figure out which client is making the request. OAuth provides a few different ways for clients to authenticate to the authorization server, and protocols that build on top of OAuth such as OpenID Connect define yet more ways (we'll take a closer look at OpenID Connect in chapter 13, but these additional methods are left as an exercise to the reader there as well). For our simple server, we're going to support the two most common methods: passing the client ID and client secret over HTTP Basic authentication, and passing them over form parameters. We're going to follow good server programming principles here and be generous with the kind of input we receive, allowing our clients to pass their credentials by either method, at their choice. We'll check the Authorization header first, since it's the preferred method in the specification, and fall back to the form parameters. The Authorization header in HTTP Basic is a base64 encoded string made by concatenating the username and password together, separated by a single colon (:) character. OAuth 2.0 tells us to use the client ID as the username and the client secret as the password, but with each of these being URL encoded first. On the server side, we have to unwrap that in reverse, so we've provided a utility function to handle the messy bits for you. Pull out the results of that function and store them as variables.

```
var auth = req.headers['authorization'];
if (auth) {
   var clientCredentials = decodeClientCredentials(auth);
   var clientId = clientCredentials.id;
   var clientSecret = clientCredentials.secret;
}
```

Next, we need to check whether the client sent its client ID and client secret values in the form body. You might be thinking that we need to check here only if there's no authorization header, but we also need to make sure that the client didn't send its ID and secret in *both* locations simultaneously, which would result in an error (and a possible security breach). If there was no error, getting the values is a simple matter of copying them out of the form input, as you can see here.

```
if (req.body.client_id) {
   if (clientId) {
        res.status(401).json({error: 'invalid_client'});
        return;
   }
```

```
   var clientId = req.body.client_id;
   var clientSecret = req.body.client_secret;
}
```

Next, we look up the client using our helper function. If we don't find a client, we return an error.

```
var client = getClient(clientId);
if (!client) {
   res.status(401).json({error: 'invalid_client'});
   return;
}
```

We also need to make sure that the client secret that came in on the wire is the same client secret that we expected for this client. If it isn't, you guessed it: we return an error.

```
if (client.client_secret != clientSecret) {
   res.status(401).json({error: 'invalid_client'});
   return;
}
```

At this point, we know that the client is valid and we can start processing the token request for real.

### 5.3.2 Processing the authorization grant request

First, we need to check the `grant_type` parameter and make sure it's one we know how to process. Our little server only supports the authorization code grant type, which is represented unsurprisingly by the value `authorization_code`. If we don't get a grant type that we know how to process, we return an error.

```
if (req.body.grant_type == 'authorization_code') {

} else {
   res.status(400).json({error: 'unsupported_grant_type'});
   return;
}
```

**Process the authorization code grant in here, see the following**

If we do get an authorization code grant, we need to pull the `code` out of the request and look it up in our code storage object that we populated in the last section. If we can't find the `code`, we return an error to the client.

```
var code = codes[req.body.code];

if (code) {

} else {
   res.status(400).json({error: 'invalid_grant'});
   return;
}
```

**Process the valid authorization code in here, see the following**

If we are able to find the `code` passed in in our code store, we need to make sure it was issued to this client. Thankfully, when we saved the `code` in the last section, we also

saved information about the request to the authorization endpoint, which includes the client ID. Compare these and, if they don't match, return an error.

```
delete codes[req.body.code];
if (code.request.client_id == clientId) {

} else {
   res.status(400).json({error: 'invalid_grant'});
   return;
}
```

**Process the valid authorization code here, see the following**

Notice that as soon as we know the code is a valid one, we remove it from storage, regardless of the rest of the processing. We do this to err on the side of caution, because a stolen authorization code presented by a bad client should be considered lost. Even if the right client shows up later with the authorization code, the authorization code won't work, as we know it has already been compromised. Next, if the clients do match, we need to generate an access token and store it so that we can look it up later.

```
var access_token = randomstring.generate();
nosql.insert({ access_token: access_token, client_id: clientId });
```

For simplicity, we're saving our token into a local, file-based NoSQL database using the Node.js `nosql` module. In a production OAuth system, you have a lot of options for what to do with your tokens. You can store your tokens into a full-scale database; for a little added security, you can store a cryptographic hash of the token value so that if your database is compromised, the tokens themselves aren't lost.[1] Alternatively, your resource server could use token introspection to look up information about the token back at the authorization server without the need for a shared database. Or, if you can't store them (or don't want to), you can use a structured format to bake all the necessary information into the token itself for the protected resource to consume later without needing to look it up. We'll cover those methods in chapter 11.

### What's in a token?

OAuth 2.0 is famously silent about what's inside an access token, and for good reason: there are many options, each with its own trade-offs that make them applicable to different use cases. Unlike previous security protocols like Kerberos, WS-Trust, and SAML, OAuth functions without the client knowing anything about what's inside the token. The authorization server and protected resource need to be able to process the token, but these components can choose whatever means they want to communicate this information with each other.

Consequently, an OAuth token could be a random string with no internal structure, as are the tokens in our exercise. If the resource server is co-located with the ***(continued)***

[1] Granted, if your security server's database gets compromised, you've got other problems to worry about.

authorization server, as in our exercise, it can look up the token value in a shared database and determine who the token was issued to, and what rights it has. Alternatively, OAuth tokens can have structure to them, like a JSON Web Token (JWT) or even a SAML assertion. These can be signed, encrypted, or both, and the client can remain oblivious of what's inside the token when they're in use. We'll go into more depth on JWTs in chapter 11.

Now that we have our token and we've stored it for later use, we can finally send it back to the client. The response from the token endpoint is a JSON object that contains the value of the access token and a `token_type` indicator that tells the client what kind of token this is and, consequently, how to use it with the protected resource. Our OAuth system uses bearer tokens, so that's what we'll tell the client. We'll look ahead toward another kind of token called Proof of Possession (PoP) in chapter 15.

```
var token_response = { access_token: access_token, token_type: 'Bearer' };
res.status(200).json(token_response);
```

With that last bit of code in place, our token endpoint handling function looks like listing 9 in appendix B.

At this point, we've got a simple but fully functioning authorization server. It can authenticate clients, prompt users for authorization, and issue randomized bearer tokens using the authorization code flow. You can try it out by starting at the OAuth client `http://localhost:9000/`, getting a token, approving it, and using it at the protected resource.

As an added exercise, add a short expiration time to the access tokens. You'll need to store their expiration time as well as send the `expires_in` response parameter back to the client. You'll also need to modify the `protectedResource.js` file to make the resource server check the token's expiration before serving its request.

## 5.4 Adding refresh token support

Now that we're able to issue access tokens, we'd like to be able to issue and honor refresh tokens as well. Refresh tokens, you'll recall from chapter 2, aren't used to access the protected resource but are instead used by the client to obtain new access tokens without requiring user intervention. Thankfully, all the work we've done to get access tokens out of our server won't go to waste, and we'll be adding to our project from the last exercise. Open up `ch-5-ex-2` and edit the `authorizationServer.js` file or, if you'd prefer, you can add on to the last exercise once it has been completed.

First, we need to issue the token. Refresh tokens are similar to bearer tokens, and they are issued alongside the access token. Inside our token endpoint function, we're going to generate and store the refresh token value right alongside our existing access token value.

```
var refresh_token = randomstring.generate();
nosql.insert({ refresh_token: refresh_token, client_id: clientId });
```

We're using the same randomized string generation function here and storing the refresh token in the same NoSQL database. However, we are storing the refresh token under a different key so that the authorization server and protected resource will be able to differentiate the tokens from each other. This is important because the refresh token is only to be used at the authorization server and the access token is only to be used at the protected resource. With the tokens generated and stored, send them both back to the client in parallel with each other:

```
var token_response = { access_token: access_token, token_type: 'Bearer',
refresh_token: req.body.refresh_token };
```

The `token_type` parameter (along with the `expires_in` and `scope` parameters, when they're sent) applies only to the access token and not the refresh token, and there are no equivalents for the refresh token. The refresh token is still allowed to expire, but since refresh tokens are intended to be fairly long lived, the client isn't given a hint about when that would happen. When a refresh token no longer works, a client has to fall back on whatever regular OAuth authorization grant it used to get the access token in the first place, such as the authorization code grant.

Now that we're issuing refresh tokens, we need to be able to respond to a request to refresh a token. In OAuth 2.0, refresh tokens are used at the token endpoint as a special kind of authorization grant. This comes with its own `grant_type` value of `refresh_token`, which we can check in the same branching code that handled our `authorization_code` grant type earlier.

```
} else if (req.body.grant_type == 'refresh_token') {
```

First, we need to look up our refresh token in our token store. We'll do that in our example code using a query against our NoSQL store, and although the specifics of this are particular to our example framework, it amounts to a simple search operation.

```
nosql.one(function(token) {
   if (token.refresh_token == req.body.refresh_token) {
        return token;
   }
}, function(err, token) {
   if (token) {          <-- We found a matching refresh token, process it with the following

   } else {
        res.status(400).json({error: 'invalid_grant'});
        return;
   }
});
```

Now we have to make sure that the token was issued to the client that authenticated at the token endpoint. If we don't make this check, then a malicious client could steal a good client's refresh token and use it to get new, completely valid (but fraudulent) access tokens for itself that look like they came from the legitimate client. We also remove the refresh token from our store, since we can assume that it's been compromised.

```
if (token.client_id != clientId) {
   nosql.remove(function(found) { return (found == token); }, function () {} );
   res.status(400).json({error: 'invalid_grant'});
   return;
}
```

Finally, if everything passes, we can create a new access token based on this refresh token, store it, and return it to the client. The response from the token endpoint is identical to the one used by other OAuth grant types. This means that a client doesn't need special processing to handle access tokens that were gotten from refresh tokens or authorization codes. We also send back the same refresh token that was used to make this request, indicating to the client that it can use that refresh token again in the future.

```
var access_token = randomstring.generate();
nosql.insert({ access_token: access_token, client_id: clientId });
var token_response = { access_token: access_token, token_type: 'Bearer',
refresh_token: token.refresh_token };
res.status(200).json(token_response);
```

The branch of your token endpoint that handles refresh tokens looks like listing 10 in appendix B.

When our client is authorized, it is given a refresh token alongside its access token. The client can now use that refresh token after the access token has been revoked or disabled, for any reason.

**Throw my tokens out!**

In addition to optionally expiring, access tokens and refresh tokens can be revoked at any time for any number of reasons. The resource owner could decide that they don't want to use the client anymore, or the authorization server could have become suspicious of a client's behavior and decide to preemptively remove all tokens issued to a client. As an added exercise, build a page on the authorization server that lets you clear out access tokens for each client in the system.

We'll be looking more into the token lifecycle, including the token revocation protocol, in chapter 11.

When a refresh token is used, the authorization server is free to issue a new refresh token to replace it. The authorization server can also decide to throw out all active access tokens that were issued to the client up until the time the refresh token was used. As an added exercise, add these capabilities to the authorization server.

## 5.5 Adding scope support

An important mechanism in OAuth 2.0 is the scope. As we discovered in chapter 2, and saw in practice in chapter 4, scopes represent a subset of access rights tied to a specific authorization delegation. To fully support scopes, we're going to have to

change a few things around the server. Open up `ch-5-ex-3` and edit the `authorizationServer.js` file, or keep working from the previous exercise once it's been completed. The `client.js` and `protectedResource.js` files can remain untouched for this exercise.

To start us off, it's common to limit which scopes each client can access at a server. This provides a first line of defense against misbehaving clients, and allows a system to limit which software can perform certain actions at a protected resource. We're going to add a new member to our client structure at the top of the file: `scope`.

```
var clients = [
   {
        "client_id": "oauth-client-1",
        "client_secret": "oauth-client-secret-1",
        "redirect_uris": ["http://localhost:9000/callback"],
        "scope": "foo bar"
   }
];
```

This member is a space-separated list of strings, each string representing a single OAuth scope value. Merely being registered like this doesn't give an OAuth client access to the things protected by that scope, as it still needs to be authorized by the resource owner.

A client can ask for a subset of its scopes during its call to the authorization using the scope parameter, which is a string containing a space-separated list of scope values. We'll need to parse that in our authorization endpoint, and we're going to turn it into an array for easier processing and store it in the `rscope` variable. Similarly, our client can optionally have a set of scopes associated with it, as we saw previously, and we'll parse that into an array as the `cscope` variable. But because `scope` is an optional parameter, we need to be a little bit careful in how we handle it, in case a value wasn't passed in.

```
var rscope = req.query.scope ? req.query.scope.split(' ') : undefined;
var cscope = client.scope ? client.scope.split(' ') : undefined;
```

By parsing things in this way, we can avoid accidentally trying to split a nonexistent value on spaces, which would otherwise make the code execution fail.

### Why a space-separated set of strings?

It may seem odd that the scope parameter is represented as a space-separated list of strings (encoded into a single string) throughout the OAuth process, especially when some parts of the process, such as the response from the token endpoint, use JSON, which has native support for arrays. You'll also notice that when we're working with scopes inside our code, we use an array of strings natively. You may even have picked up on the fact that this encoding means that scope values can't have spaces in them (because that's the delimiter character). Why bother with the odd encoding?
***(continued)***

As it turns out, HTTP forms and query strings don't have a good way to represent complex structures such as arrays and objects, and OAuth needs to use query parameters to pass values through the front channel. To get anything into this space, it needs to be encoded in some fashion. Although there are a few relatively common hacks such as serializing a JSON array as a string or repeating a parameter name, the OAuth working group decided that it would be much simpler for client developers to concatenate scope values, separated by a space character, into a single string. The space was chosen as a separator to allow for a more natural separator between URIs, which some systems use for their scope values.

We then need to make sure that a client isn't asking for any scopes beyond its means. We can do this with a simple comparison between the requested scopes and the client's registered scopes (done here using the Underscore library's set difference function).

```
if (__.difference(rscope, cscope).length > 0) {
   var urlParsed = buildUrl(req.query.redirect_uri, {
        error: 'invalid_scope'
   });
   res.redirect(urlParsed);
   return;
}
```

We're also going to amend our call to the template for the user approval page by passing in the `rscope` value. This will let us render a set of checkboxes, allowing the user to select exactly which scopes they're approving the client for. In this way, the client could get back a token less powerful than the one it asked for, but that's up to the authorization server and, in our case, the resource owner. If the client isn't happy with the scopes it was granted, it can always try to ask the user again. In practice, this annoys the user, and clients are best advised to ask for only the scopes they need to function to avoid this situation arising.

```
res.render('approve', {client: client, reqid: reqid, scope: rscope});
```

Inside our page, we've got a small block that will loop through the scopes and render a checkbox for each one as part of the form. We've already provided the code to do this, but if you want, open the `approve.html` page to see for yourself what that code looks like.

```
<% if (scope) { %>
<p>The client is requesting access to the following:</p>
<ul>
<% _.each(scope, function(s) { %>
   <li><input type="checkbox" name="scope_<%- s %>" id="scope_<%- s %>"
   checked="checked"> <label for="scope_<%- s %>"><%- s %></label></li>
<% }); %>
</ul>
<% } %>
```

We check all the boxes initially because the client probably has its own reasons for asking for those permissions, and most users are likely to leave the page in a default state. However, we do want to give resource owners the option of removing some of these privileges by unchecking the checkboxes.

Now we're going to look inside the function that handles the processing of the approval page. Remember, it starts like this:

```
app.post('/approve', function(req, res) {
```

Since the form template uniquely labels all of the checkboxes with a `scope_` prefix and the scope value, we can figure out which ones have been left checked, and therefore which scopes have been allowed by the resource owner, by looking at the incoming data from the form. We're going to use a couple of Underscore functions to help us do this processing more cleanly, but it could also be done with a `for` loop if you prefer. We've packaged this into a utility function and included it for you.

```
var getScopesFromForm = function(body) {
   return __.filter(__.keys(body), function(s) { return
   __.string.startsWith(s, 'scope_'); })
                        .map(function(s) { return
   s.slice('scope_'.length); });
};
```

Now that we have our list of approved scopes, we once again need to make sure that it doesn't exceed what the client is authorized for. "Wait a second," you may be asking, "didn't we already check that in the last step?" We did, but the form we rendered to the browser or its resulting POST could have been manipulated by the user or by code running inside the browser. New scopes could have been injected that the client didn't ask for, and potentially isn't authorized for. Besides, it's always a good idea for a server to validate all of its inputs, wherever possible.

```
var rscope = getScopesFromForm(req.body);
var client = getClient(query.client_id);
var cscope = client.scope ? client.scope.split(' ') : undefined;
if (__.difference(rscope, cscope).length > 0) {
   var urlParsed = buildUrl(query.redirect_uri, {
        error: 'invalid_scope'
   });
   res.redirect(urlParsed);
   return;
}
```

Now we need to store these scopes along with our generated authorization code so that they can be picked up again at the token endpoint. You'll notice that by using this technique, we can hang all kinds of arbitrary information off the authorization code, which can help for advanced processing techniques.

```
codes[code] = { request: query, scope: rscope };
```

Next, we need to edit the handler for the token endpoint. Recall that it starts with this:

```
app.post("/token", function(req, res){
```

In here, we need to pull these scopes back out from the original approval and apply them to our generated tokens. Since they were stored with the authorization code object, we can grab them from there and put them into our tokens.

```
nosql.insert({ access_token: access_token, client_id: clientId, scope:
code.scope });
nosql.insert({ refresh_token: refresh_token, client_id: clientId, scope:
code.scope });
```

Finally, we can tell the client about the scope that the token was issued with in the response from the token endpoint. To be consistent with the space-separated formatting used during the request, format our scope array back to a string as we add it to the response JSON object.

```
var token_response = { access_token: access_token, token_type: 'Bearer',
refresh_token: refresh_token, scope: code.scope.join(' ') };
```

Now our authorization server can handle requests for scoped tokens, allowing the user to override which scopes are issued to the client. This lets our protected resources split up access more finely, and it lets our clients ask for only the access they need.

Refresh token requests are allowed to specify a subset of the scopes that the refresh token was issued with to tie to the new access token. This lets a client use its refresh token to ask for new access tokens that are strictly less powerful than the full set of rights it has been granted, which honors the security principle of least privilege. As an additional exercise, add this down-scoping support to the `refresh_token` grant type in the token endpoint handler function. We've left basic refresh token support in the server, but you'll need to hack the token endpoint to parse, validate, and attach the scopes appropriately.

## 5.6 Summary

The OAuth authorization server is arguably the most complex portion of the OAuth system.

- Handling the front channel and back channel responses communication requires different techniques, even for similar requests and responses.
- The authorization code flow requires tracking data across multiple steps, resulting in an access token.
- Many potential locations for attack against the authorization server exist, all of which need to be appropriately mitigated.
- Refresh tokens are issued alongside access tokens and can be used to generate new access tokens without user involvement.
- Scopes limit the rights of an access token.

Now that you've seen how all of the different components in the OAuth system work with the most canonical setup, let's take a look at a few of the other options and how the whole system fits together in the real world.